[2022 Operating System] Final Report

Group members: 407410003 常亦德、407410065 王宸潤、408410042 林靖紳

Abstract

Parallel or multiple threads and processes often suffer from synchronization problems. In the Linux kernel, the lock strategy is mainly divided into two parts: spin locks and blocking (sleep) locks. We learned about blocking strategy in the course. Therefore, this paper describes one of a locking strategy: spinning locks. We start with why locks are needed, including a brief description of race conditions, critical sections, atomic operations, and busy waiting. We then implemented some well-known spinlock algorithms, analyzed its efficacy, and compared them with Mutex and Semaphore.

Introduction

As the number of multi-core systems increases, more processes or threads execute simultaneously on different cores. When multiple processes execute the same code or access same memory or any shared variable in this situation, the output or the value of the shared variables will probably be wrong; however, all the processes doing the race return that their output value are correct. This condition is known as a race condition.

A race condition may occur inside a critical section, where only one process is allowed to access the code segment at a time, when the result of multiple threads execution differs depending on the executing order.

The critical section contains shared variables that need to be sychronized to maintain the consistency of data variables. Therefore, critical section problem means that we should designing a way to access shared resources for cooperative processes without creating data inconsistencies.

Atomic operations are used to manipulate locks; when a processor is accessing a data structure, the lock is busy, so other processors requesting access must wait, also calles as "busy waiting."

For small critical sections, spinning for a lock to be released is more efficient than relinquishing the processor to do other works; however, its overhead is spin-waiting will slow down other processors by consuming communication bandwidth.

We will devided our works into three part:

- Implementation
    - Implement spinlocks in C language
    - Design a testing structure for analyzing their efficacy
- Analysis
    - Performance Evaluation
        - Arrange critical section size
        - Arrange remainder section size
        - Oversubscribe problem
    - Fairness Evaluation
        - Count the number of times each thread enters the critical section
- Conclusion
    - Show our experimental results
    - Point out the pros and cons of spinlock, mutex, and semaphore

Implementation

In the spin lock part, we have implemented some famous spin lock algorithms, including the pthread_spin_lock we wrote, the pthread_spin_lock function in the C library, ticket lock, MCS spin lock; for mutex and semaphore, we choose pthread_mutex and binary_semaphore is to fix the number of threads that can enter the critical section to 1.

Locking programs

Inuse Only

Figure 1 shows the concept of threads exchange locks in a spinlock. Inuse only is an algorithm we wrote in Figure 2 based on the method of GNU Pthread's spinlock. We used a variable called Inuse to represent the lock state. A thread intending to enter a critical section checks the value of Inuse until it equals 0. When a thread leaves the critical section, it sets Inuse to 0. Only the first thread to observe that Inuse is 0 can enter the critical section. Therefore, the closer the thread is to releasing the lock, the more likely the thread will enter the critical section.

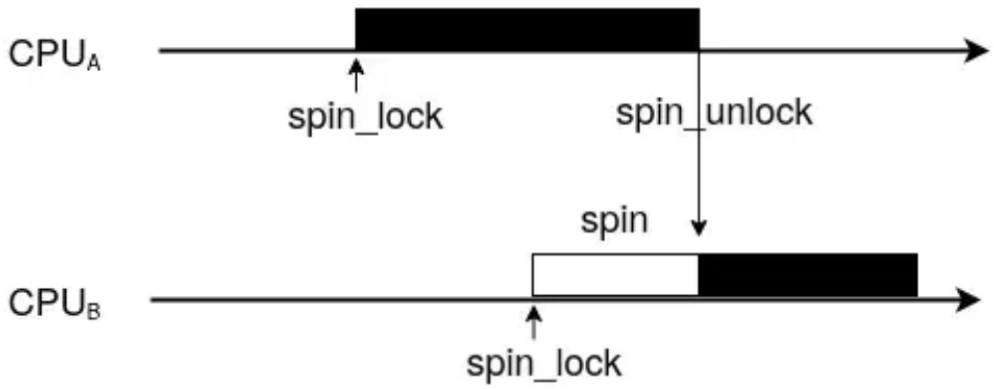


Figure 1

```
// InUse is defined in atomic_int in public.h and it is initialized to 0
void spin_init(){

}
void spin_lock(){
    while(1){
        while(InUse!=0){
            asm("pause");
            //thrd_yield();
        }
        if(unlikely(InUse==0)){
            if(atomic_compare_exchange_weak(&InUse, &flag, 1)){
                return ;
            }
        }
        flag =0;
    }
}
void spin_unlock(){
    InUse=0;
}
```

Figure 2

Pthread Spin

We used the real GNU Pthreads spinlock function in Figure 3.

```
2  pthread_spinlock_t spin;
3
4  void spin_init(){
5      pthread_spin_init(&spin, PTHREAD_PROCESS_PRIVATE);
6  }
7  void spin_lock(){
8      pthread_spin_lock(&spin);
9  }
10 void spin_unlock(){
11     pthread_spin_unlock(&spin);
12 }
```

Figure 3

Ticket Spinlock

Figure 4 shows the concept of a ticket spinlock. This method makes each thread waiting to enter the critical section have a "ticket number". The thread waits until the "service number" is equal to its ticket number before entering the critical section. All threads waiting in the loop constantly query the value of "grant" by consuming limited NoC bandwidth. We implement the above method in Figure 5.

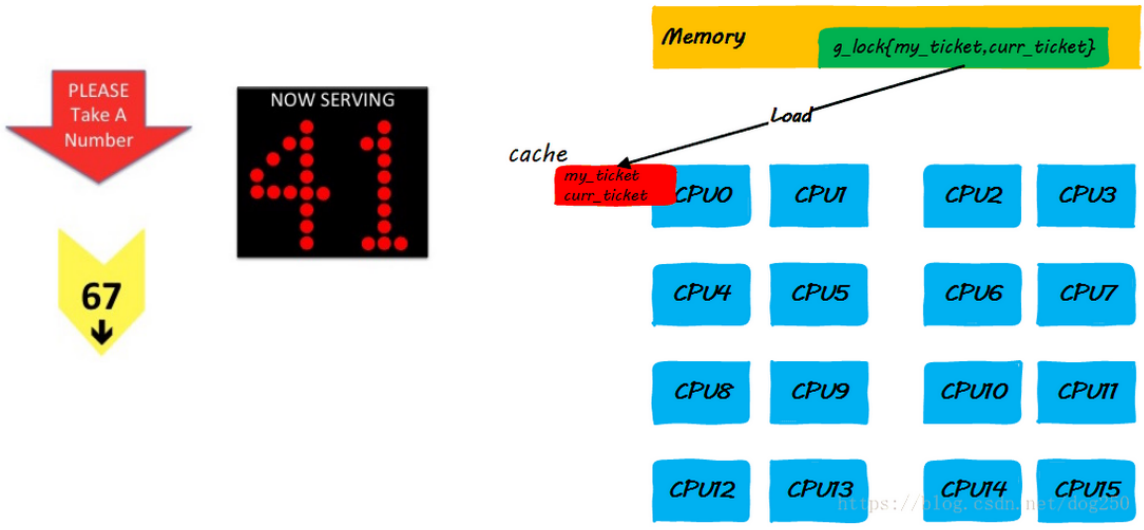


Figure 4

```
3  atomic_llong global_ticket={0};
4  atomic_llong service_ticket={0};
5
6  void spin_init(){
7      atomic_store_explicit(&global_ticket, 0, memory_order_release);
8      atomic_store_explicit(&service_ticket, 0, memory_order_release);
9  }
10
11 void spin_lock(){
12     long long int local_ticket;
13     local_ticket = atomic_fetch_add_explicit(&global_ticket, 1, memory_order_relaxed);
14     while(local_ticket != service_ticket) {
15         //asm("pause");
16         thrd_yield();
17     }
18 }
19
20 void spin_unlock(){
21     atomic_fetch_add_explicit(&service_ticket, 1, memory_order_relaxed);
22 }
```

Figure 5

MCS Spinlock

Figure 6 indicates the concept of a MCS spinlock and we implemented MCS spinlock in Figure 7 and Figure 8. All threads waiting to enter the critical section are queued in the linked list. When a thread leaves the critical section, it just needs to set the "locked" variable of the next thread to false. The "locked" variable is like the thread's waiting flag, indicating whether the thread can enter the critical section.

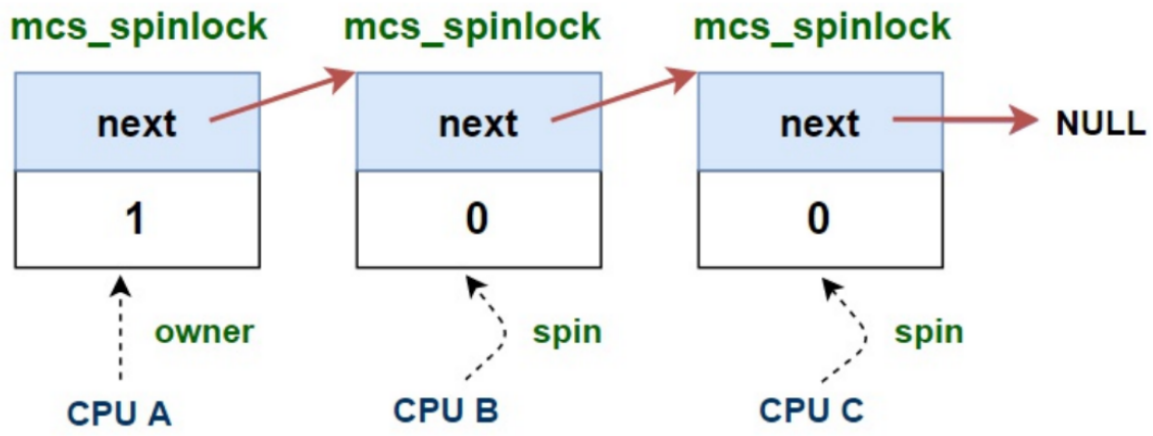


Figure 6

```
2  #define mcs_null (struct mcs_spinlock*)NULL
3
4  _Atomic struct mcs_spinlock* mcs_node;
5
6  void spin_init(){
7      mcs_node = malloc(sizeof(struct mcs_spinlock));
8      mcs_node->next = mcs_null;
9      mcs_node->locked = 0;
10 }
```

Figure 7

```
12 void mcs_spin_lock(mcs_spinlock* node){
13
14     atomic_init(&(node->next), mcs_null);
15     mcs_spinlock* prev = __atomic_exchange_n(&(mcs_node->next), node, 0);
16
17     if(prev != mcs_null){
18         atomic_init(&(node->locked), 1);
19         atomic_store_explicit(&(prev->next), node, memory_order_release);
20         while(atomic_load_explicit(&(node->locked), memory_order_acquire)) {
21             //asm("pause");
22             thrd_yield();
23         }
24
25     }
26
27 }
28
29 void mcs_spin_unlock(mcs_spinlock* node){
30     mcs_spinlock* with_lock = atomic_load_explicit(&(node->next), memory_order_acquire);
31     if(with_lock == mcs_null){
32         mcs_spinlock* prev = node;
33         if(atomic_compare_exchange_weak(&(mcs_node->next), &prev, mcs_null)) return ;
34
35         while((with_lock = atomic_load_explicit(&(node->next), memory_order_acquire)) == mcs_null){
36             //asm("pause");
37             thrd_yield();
38         }
39     }
40     atomic_store_explicit(&(with_lock->locked), 0, memory_order_release);
41 }
```

Figure 8

Pthread Mutex

We used pthread_mutex in GNU pthread mutex lock function as shown in Figure 9. If the mutex is already locked, the calling thread blocks until it acquires the mutex. When the function returns, the mutex object is locked and owned by the calling thread.

```
4  void spin_init(){
5      pthread_mutex_init(&mutexlock, PTHREAD_PROCESS_PRIVATE);
6  }
7  void spin_lock(){
8      pthread_mutex_lock(&mutexlock);
9  }
10 void spin_unlock(){
11     pthread_mutex_unlock(&mutexlock);
12 }
```

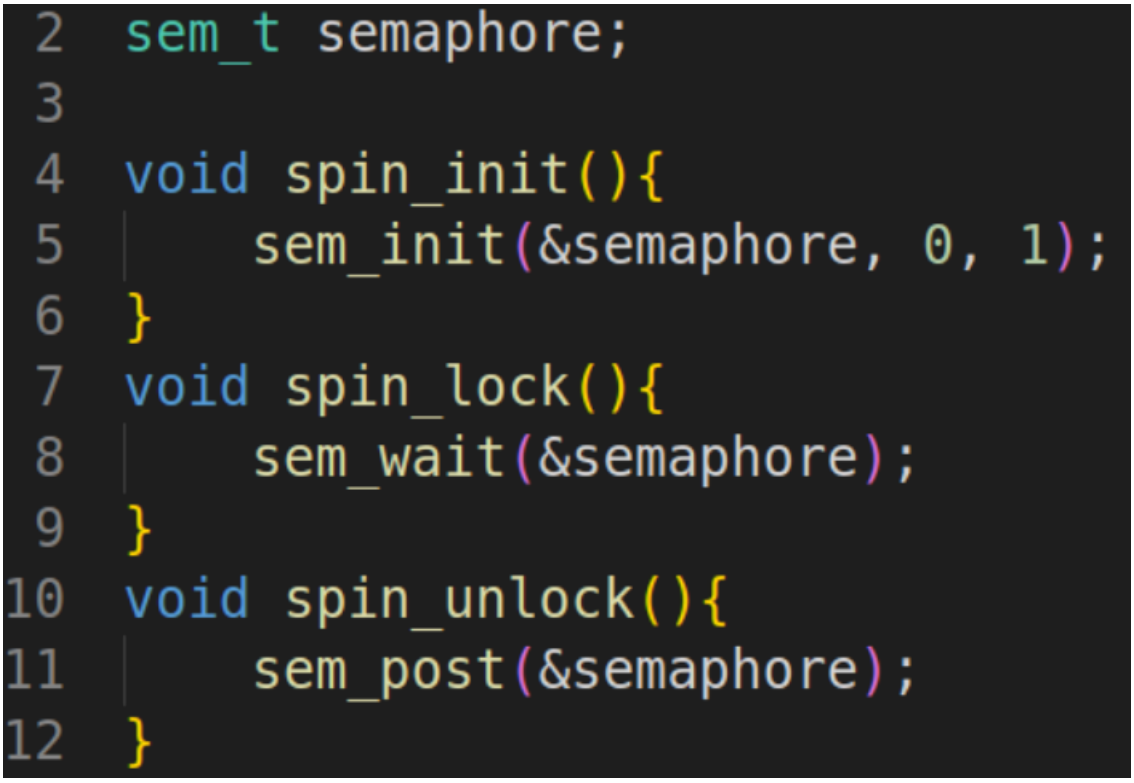
Figure 9

Binary Semaphore

We used sem_lock() function in <semaphore.h> library to implement binary semaphore in Figure 10. The reason why we select the binary semaphore as the test program is to fix the number of threads that can enter the critical section as 1, so as to ensure tests are fair when comparing against spinlocks.

The sem_wait() function decrement semaphore when its value is greater than zero. If the value of the semaphore is zero, then the current thread will block until the semaphore's value becomes greater than zero.

If the resulting value is greater than zero and if there is a thread waiting on the semaphore, the sem_post function makes the waiting thread decrement the semaphore value by one and continues running.



```
2  sem_t semaphore;
3
4  void spin_init(){
5      sem_init(&semaphore, 0, 1);
6  }
7  void spin_lock(){
8      sem_wait(&semaphore);
9  }
10 void spin_unlock(){
11     sem_post(&semaphore);
12 }
```

Figure 10

Testing Programs - Microbenchmarks

We analyze each lock method in a quantitative manner through a controlled microbenchmark. First, we bind each thread to a core, as shown in Figure 11. To ensure the fairness of the test, and stipulate that each core can only process one thread.

```
21 void *dothread(void *arg){
22     int num = atomic_fetch_add_explicit(&counter, 1, memory_order_release);
23     num %= Num_core;
24     cpu_set_t cpuset;
25     CPU_ZERO(&cpuset);
26     CPU_SET(num, &cpuset);
27     sched_setaffinity(0, sizeof(cpuset), &cpuset);
28
29     if(num!=sched_getcpu()){
30         printf("false.\n");
31     }//printf("%d ",num);
32
33     double rnd;
34     rnd=(double)(rand()%31)*0.01+0.85;
35     rnd *= nCS_size;
```

Figure 11

Critical section

Next, program executes the critical section shown in Figure 12 and Figure 13. In the while loop, the lock thread requests to enter the critical section. After thread enters the critical section, *user* , announced as a pointer array represents a shared data to be read and written.

Here, we design two different loads to measure the efficacy of each lock in the critical section in Figure 12 and Figure 13.

Arithmetic

Figure 12 shows the arithmetic operation on shared data. We use a *for* loop to execute *user_num* times, incrementing the shared data by 1 each time. We also check that the data is correct by checking the value of each element in *user_num*.

Below are the meaning of variables:

- *thread_cs_counter*
  - Type:
    - It is an array of integers whose size is the number of cores
  - Function:
    - Count the number of times each thread enters the critical section
- rounds
  - Type:
    - It is an integer variable
  - Function:
    - Count the total number of locks and unlocks during our test time
- error
  - Type:
    - It is an integer variable and initial as 0;
  - Function:
    - Check if only one thread enters critical section

```
42      // critical section
43      while(running_flag=1){
44          if(time_diff(start, current)> run_seconds*1000000000.0) break;
45          //clock_gettime(CLOCK_MONOTONIC, &lock_begin);
46          spin_lock() ;
47          //clock_gettime(CLOCK_MONOTONIC, &lock_end);
48          //printf("%d\n", sched_getcpu());
49
50          thread_cs_counter[num]++;
51
52          error++;
53          if(error!=1){
54              printf("ERROR in CS of %d: %d\n",error,num);
55          }
56          for(int i=0;i<user_num;i++){
57              (*(user+i))+=1;
58          }
59          rounds++;
60          error--;
61
62          //clock_gettime(CLOCK_MONOTONIC, &unlock_begin);
63          spin_unlock() ; //US
```

Figure 12

File I/O

Figure 13 shows the file input/output operation on shared data. We write data to the test file using the *fprintf* functions from the C library.

```
42      // critical section
43      while(running_flag=1){
44          if(time_diff(start, current)> run_seconds*1000000000.0) break;
45          //clock_gettime(CLOCK_MONOTONIC, &lock_begin);
46          spin_lock() ;
47          //clock_gettime(CLOCK_MONOTONIC, &lock_end);
48          //printf("%d\n", sched_getcpu());
49
50          thread_cs_counter[num]++;
51
52          error++;
53          if(error!=1){
54              printf("ERROR in CS of %d: %d\n",error,num);
55          }
56          for(int i=0;i<user_num;i++){
57              (*(user+i))+=1;
58              fprintf(test_file, "My user_num is %d\n", i);
59          }
60          rounds++;
61          error--;
62
63          //clock_gettime(CLOCK_MONOTONIC, &unlock_begin);
64          spin_unlock() ; //US
```

Figure 13

Remainder section

After the thread leaves the critical section, *clock_gettime()* defined in the POSIX standard is called in the remaining section (also known as the non-critical section), as shown in Figure 14. The *do while* loop executes until the remaining running time exceeds the value of $nCS_size±15%$, which is defined in Figure 15. The random variable is used to simulate different loads of the program.

```
68          clock_gettime (CLOCK_MONOTONIC, &rs_start) ; //nCS
69
70          do{
71              clock_gettime(CLOCK_MONOTONIC, &current);
72              dif_naro = time_diff(rs_start, current);
73          }while(dif_naro < rnd);
```

Figure 14

```
33          double rnd;
34          rnd=(double)(rand()%31)*0.01+0.85;
35          rnd *= nCS_size;
36
```

Figure 15

Performance Evaluation

Critical Section

The critical section size indicates the number of times the *for loop* is executed in the critical section, for example *for(0 to 100)* means the critical section size = 100.

Arithmetic

Figure 16 shows the raw data for locks per second for different critical section sizes under arithmetic operation. We can see that when the critical section exceeds 10000, spinlocks drops sharply; however, take semaphore and mutex compare with ticket and mcs spinlocks, they win out slightly.

Figure 17 is a bar chart of a portion of Figure 16 to make it easier to see the gap

```
user_num = [100, 1000, 10000, 100000, 1000000]
inuse_only = [76.2, 23.4, 3.4, 0.36, 0.035]
plock = [43.8, 25.7, 3.5, 0.31, 0.027]
ticket = [39.1, 15.7, 2.1, 0.22, 0.024]
mcs = [41.3, 15.2, 2.1, 0.22, 0.024]
mutex = [30.2, 13.1, 2.1, 0.23, 0.024]
semaphore = [29.4, 13.4, 2.1, 0.24, 0.024]
```

Figure 16

Locks per second under different loads

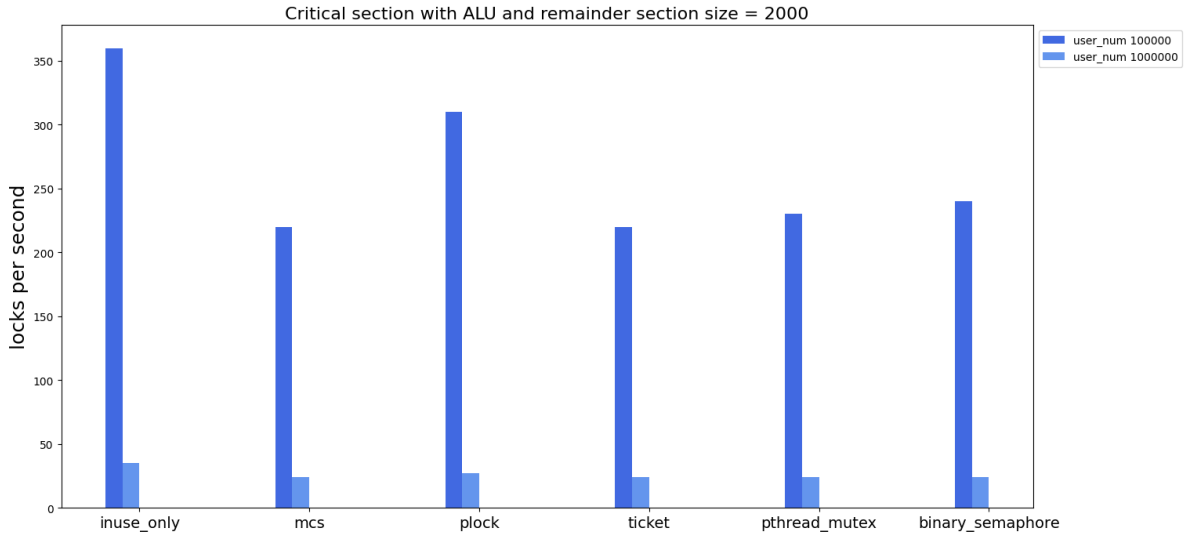


Figure 17

File I/O

Figure 18 shows the raw data for locks per second for different critical section sizes under file operation. Semaphore and Mutexe far outperform spinlocks when the critical section size exceeds 10000 in Figure 19.

```
user_num = [100, 1000, 10000, 100000, 1000000]
inuse_only = [44.4, 5.3, 0.52, 0.053, 0.0059]
plock = [77.9, 9.6, 0.95, 0.092, 0.0099]
ticket = [40.1, 5.3, 0.52, 0.052, 0.0058]
mcs = [39.7, 5.3, 0.52, 0.0052, 0.0060]
mutex = [54.5, 9.9, 1.4, 0.13, 0.013]
semaphore = [50.5, 8.9, 1.38, 0.14, 0.013]
```

Figure 19

Locks per second under different loads

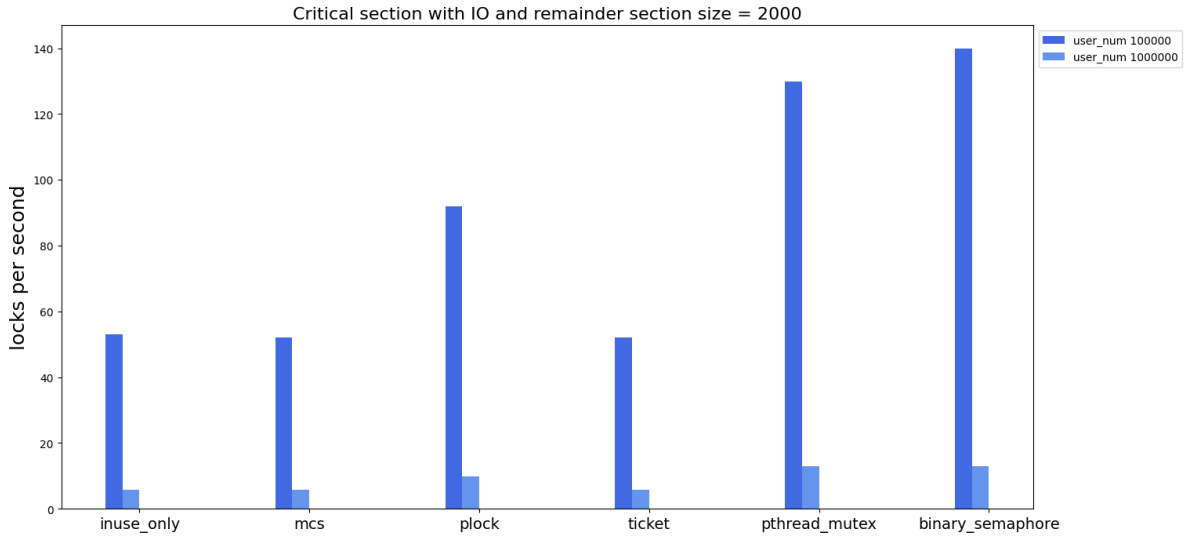


Figure 20

Remainder Section

The remainder section size indicates how long the thread will wait after leaving the critical section.

The higher the remainder section size number, the longer the waiting time. That is, it is not a competitive state, we call it a low load condition; on the contrary, if the size of the remainder section size is small, we call it a high load condition.

Arithmetic

We fix the critical section size at 100, and then measure the number of locking times under different remainder section size in Figure 21. We can see that semaphore and mutex are inefficient under arithmetic operations.

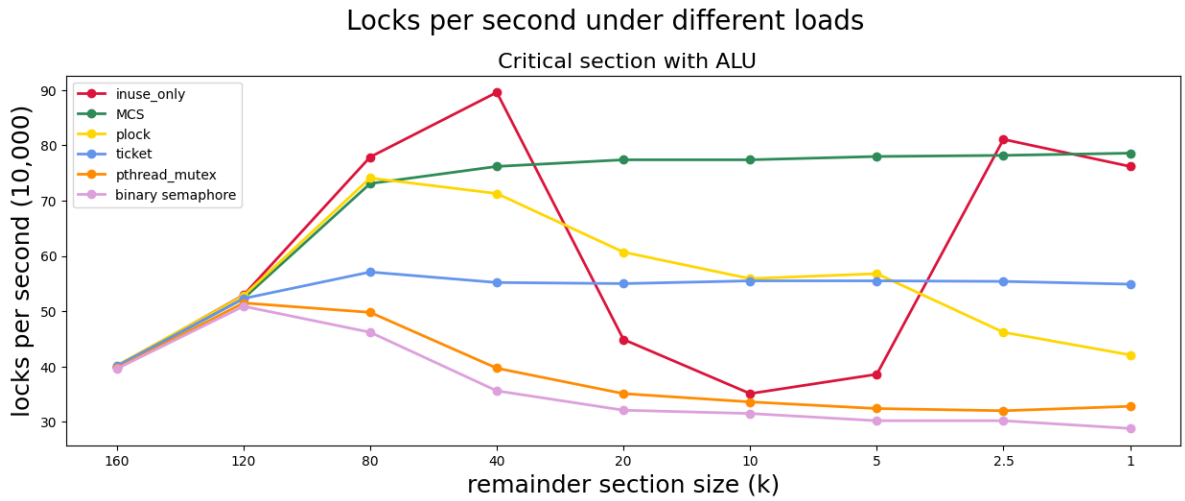


Figure 21

File I/O

We fix the critical section size at 100, and then measure the number of locking times under different remainder section size in Figure 22. We can see that the perforance of semaphore and mutex slowly increases in the case of high load condition.

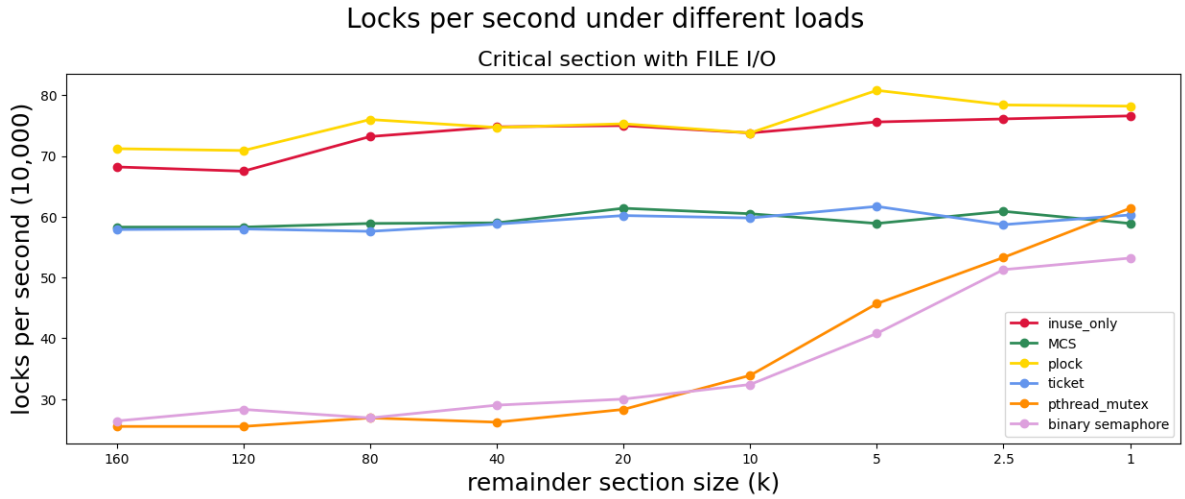


Figure 22

Oversubscribe

In the case of oversubscription, two factors affect performance. One is whether the thread with the lock is scheduled, and the other is whether the algorithm specifies whether the next thread entering the critical section is scheduled.

Arithmetic

We fix the remainder section size at 2000, and the critical section size at 100. Figure 23 shows the results. We can see that mutex and semaphore have a higher performance than most spinlocks when the numebr of threads is high.

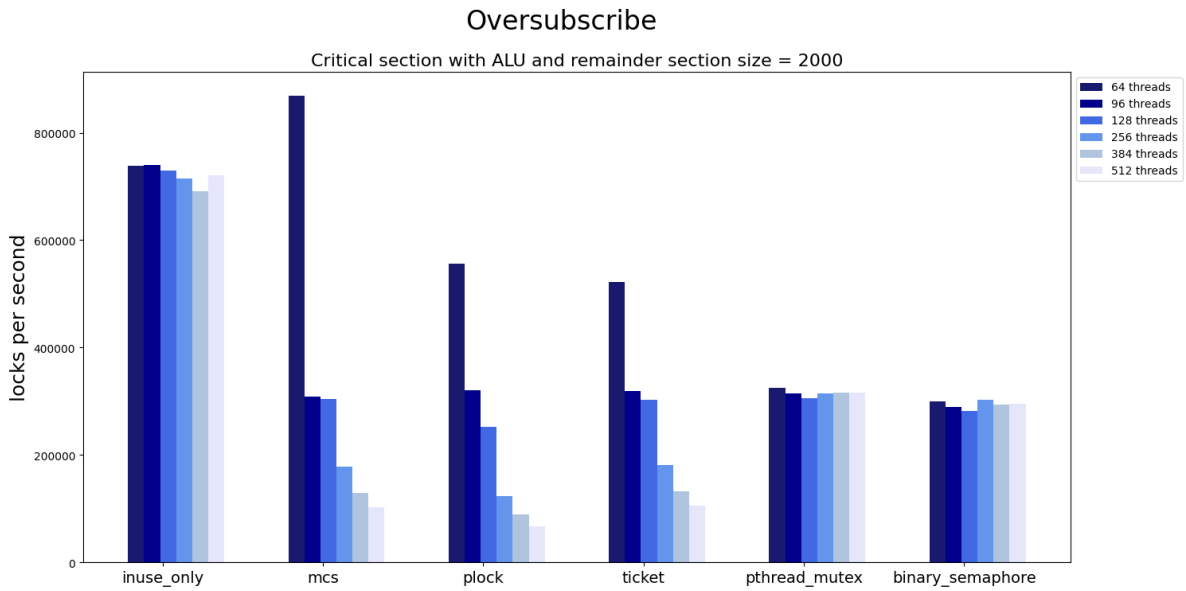


Figure 23

File I/O

Here we can see that in the case of file operations, when the number of threads is high, the spinlock drops very quickly; however, the mutex have an upward trend in Figure 24.

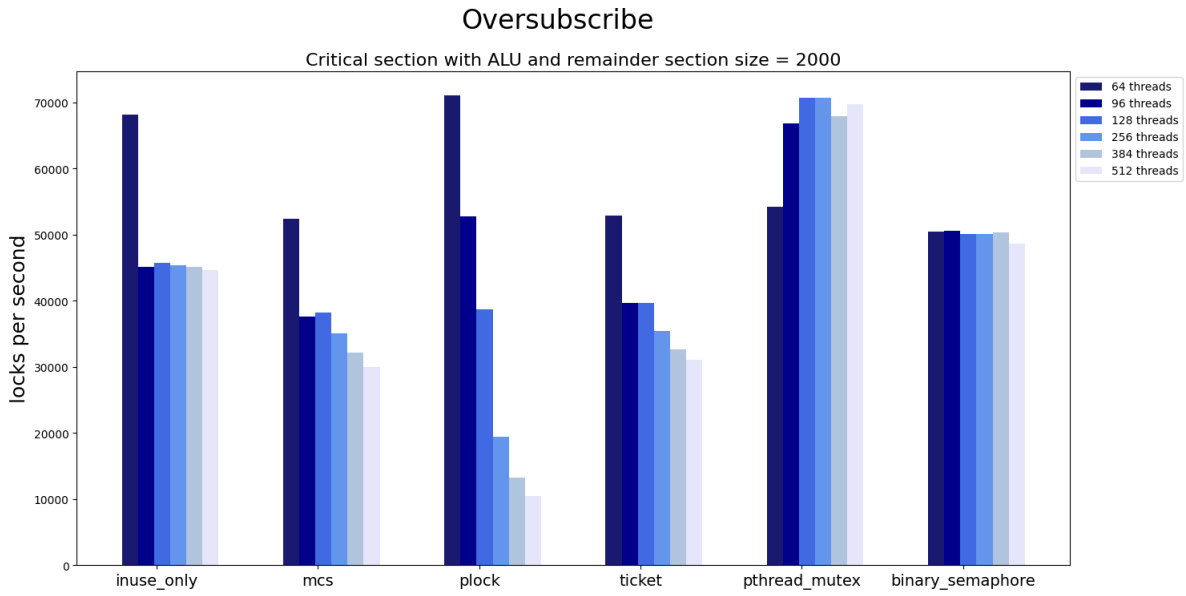


Figure 24

Figure 25 and Figure 26 are a performance comparison of arithmetic and file I/O operations for the same critical section size and remainder section size.

We can observe that when in the file I/O condition, mutex and semaphore have better performance than spinlocks. When in the arithmetic operation, spinlocks have better performance than mutex and semaphore.

Arithmetic

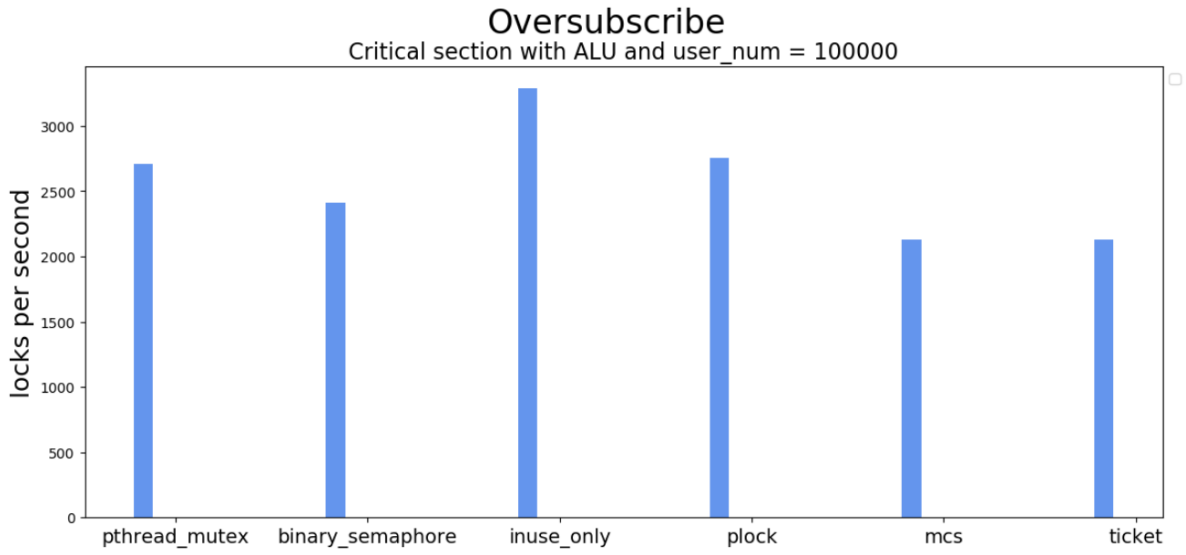


Figure 25

File I/O

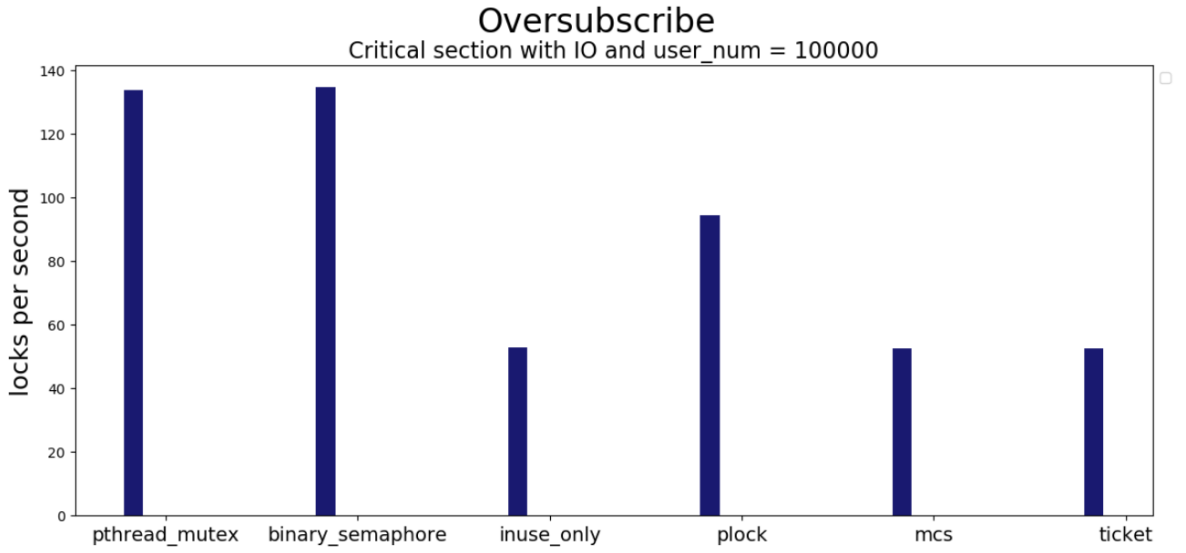


Figure 26

Fairness Evaluation

Fairness is also an important issue with locks. If a lock is losing it fairness, it can cause CPU starvation issues and degrade the overall CPU performance.

Therefore, in Figure 27 and Figure 28, we monitor the fairness of locks by measuring the number of times each thread enters a critical section.

Arithmetic

We can see that the *inuse_only* algorithm and *pthread_spin_lock* are extremely unfair; however the ticket and mcs spinlocks are fair to each thread.

The mutex and semaphore are a bit unfair, but not as extreme as pthread_spin_lock.

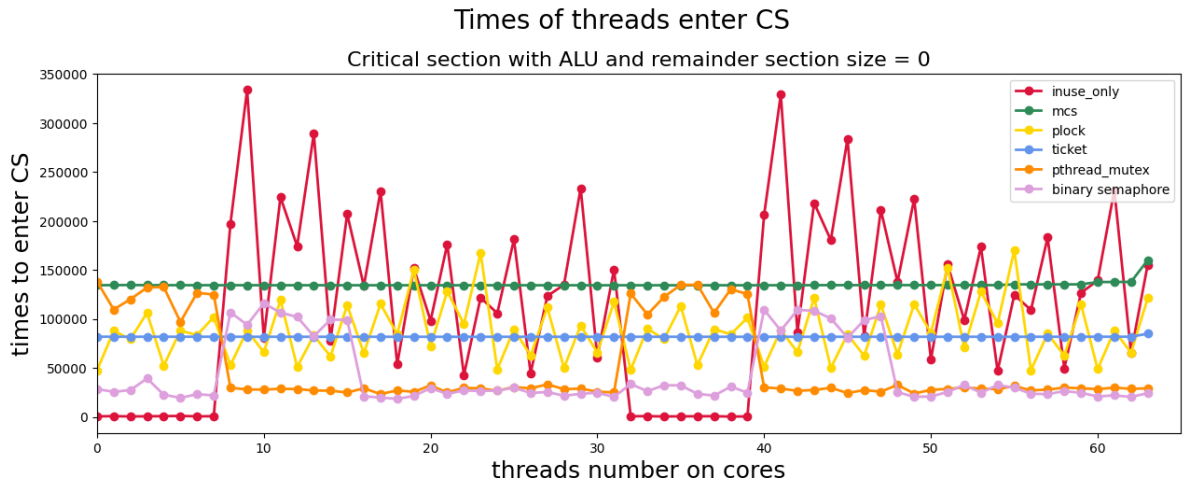


Figure 27

File I/O

We can see under file I/O condition, all locks except *inuse_only* and *pthread_spin_lock* are all fair in Figure 28.

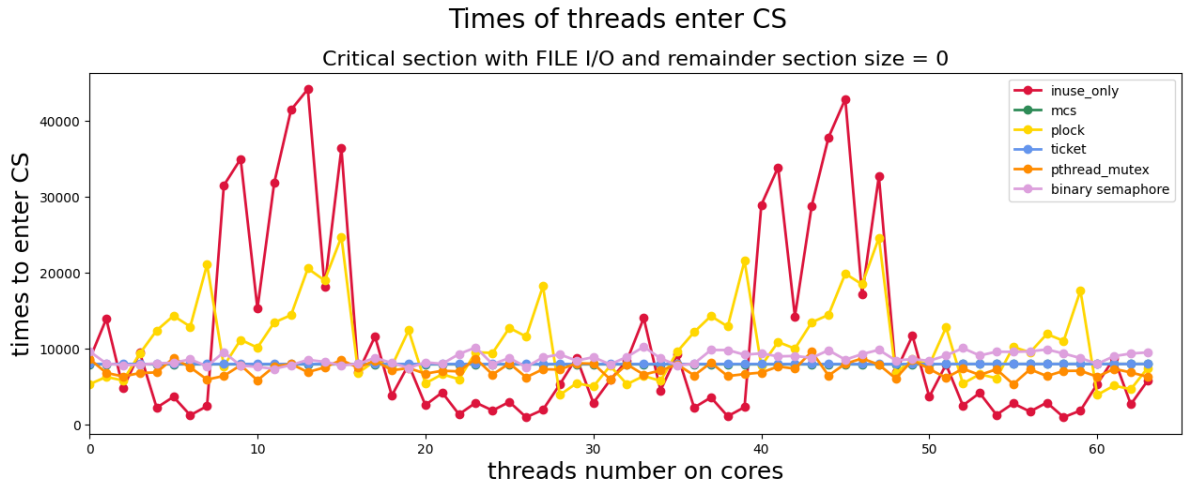


Figure 28

Conclusion

Mutex and Semaphore v.s. Spinlocks

According to the above experimental results, we can conclude that mutex and sempahore are more stable, and perform better in the case of oversubscribe and file I/O. On the contrary, spinlocks are more efficient, and perform better under arithmetic operation. However, it doesn't do well in handling oversubscription issues and long time period waiting condition.

Arithmetic

In Figure 21, mutex and semaphore perform poorly under arithmetic operations regardless of light or heavy load (small or high remainder section size).

In Figure 23, as the number of threads increases, the performance of spinlocks drops significantly, but mutex and sempahore maintain stable performance.

In Figure 25, when critical section is doing lots of arithmetic operations, the performance of all

lock methods is similar, and inuse_only and plock are slightly higher.

In the case of arithmetic, the efficiency of mutex and semaphore is worse than that of spinlock but the performance is stable in the case of oversubscribed. We think it is because the arithmetic operation is very fast and the waiting time is not much, which means that the cost of busy-waiting is very small.

File I/O

In Figure 22, when the remainder section size is < 5000, the mutex and semaphore climb sharply, but are still lower than the spinlocks.

In Figure 24, mutex and semaphore are not only stable but also effective, when the number of threads = 128 (2 times of the number of CPU thread), all spinlocks perform worse than mutex and semaphore.

In Figure 26, when the critical section is doing a lot of file I/O operations, unlike Figure 25, mutex and semaphore are stronger than other spinlocks.

In the case of file I/O, the context switching of file input and output are more obvious, busy waiting is a waste of time, resulting in spinlock performance lower than mutex and semaphore.

Working Division:

- 報告主要擔當與編寫 : 林靖紳
- 雜事分擔 : 常亦德
- 榮譽組員 : 王宸潤